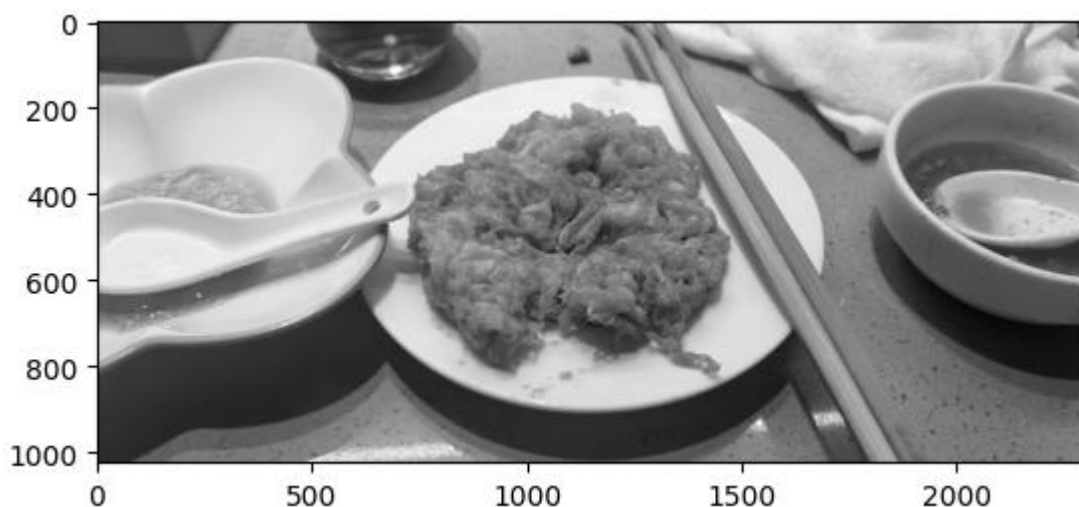


1. 读入一张图片并做灰度处理，初期效果图：

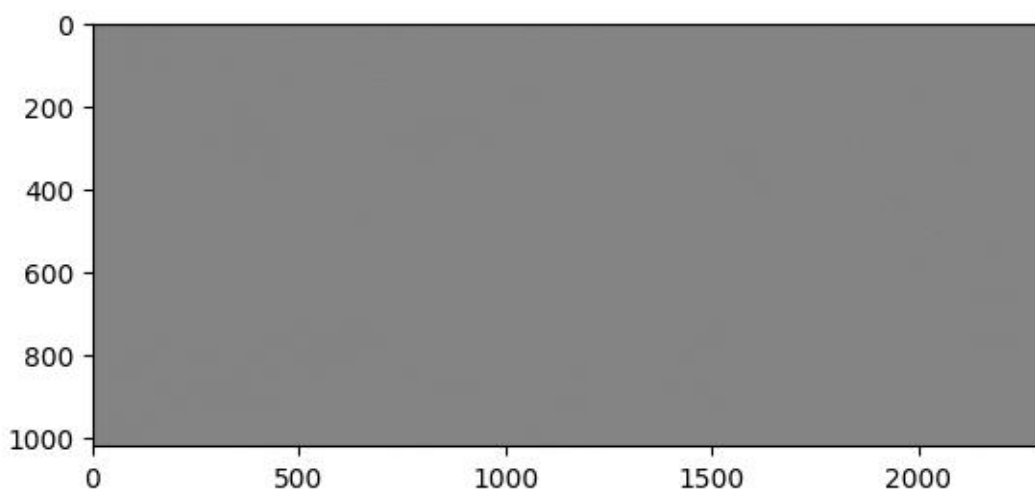


2. 定义算子对其进行轮廓检测：

```
in_ch = 1
conv = nn.Conv2d(in_channels=in_ch, out_channels=2, kernel_size=3,
                  stride=1, padding=1, bias=False)
```

参数设置：kernel_size=3 能在不增多参数的前提下有效捕捉局部边缘与纹理，是卷积网络中比较常用的核尺寸；stride=1 + padding=1 让输出与输入空间尺寸一致，便于后续网络对齐与堆叠；out_channels=2 控制该层只提取两组基础特征，**参数量小、计算开销低**，适合做早期或示例性特征提取；bias=False 在通常会接 BatchNorm 或为减少冗余参数的场景下合理。

问题解决：第一次按照官网指示进行轮廓处理时结果为全灰图，检测效果很差，如下图：



原因：原始输出范围可能是 $[-0.5, 0.5]$ 。显示时负值被截断为 0，正值被压缩。导致大部分像素集中

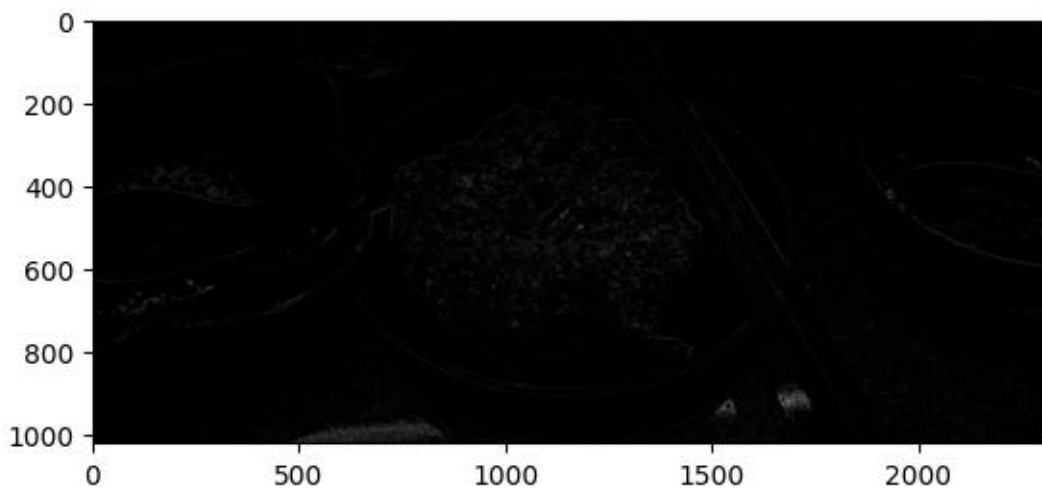
在狭窄的灰度范围内。

(1) 后采用归一化改进策略:

利用 `np.abs()` 将负边缘转换为正边缘, 保留所有边缘信息。/ `edge1.max()`: 将数据线性拉伸到 `[0,1]` 范围, 增强对比度。

```
# 归一化处理
edge1 = np.abs(edge1)
edge1 = edge1 / edge1.max()
```

处理后效果:

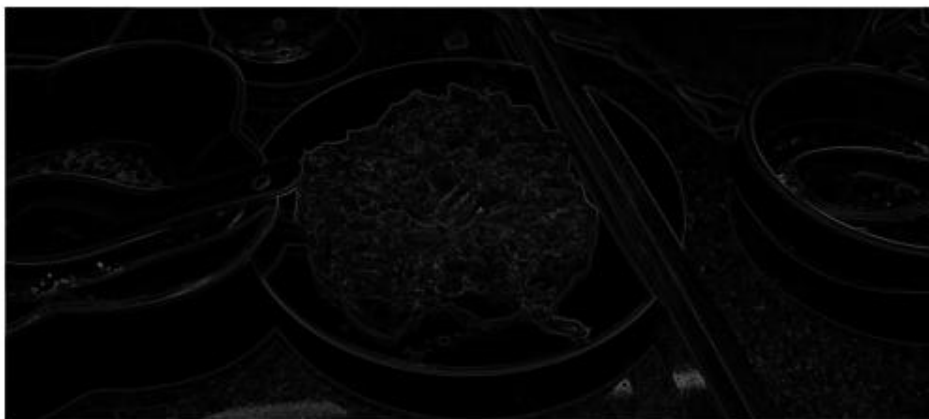


有了些许提升, 至少是有轮廓出现。

(2) 进一步优化, 利用 Sobel 算子:

拉普拉斯算子作为二阶导数对边缘产生包含正负值的双极性响应, 直接可视化时负值被截断导致对比度丧失而呈现灰色; 而 Sobel 算子通过分别计算水平与垂直方向的一阶梯度, 再融合为梯度幅值, 自然生成全正值的边缘强度图, 从而通过归一化即可充分利用显示动态范围, 呈现出清晰可见的边缘结构。

效果图:



可见有明显提升。

实验 1-2: 熟悉并实现 `torch.nn.MaxPool2d()`

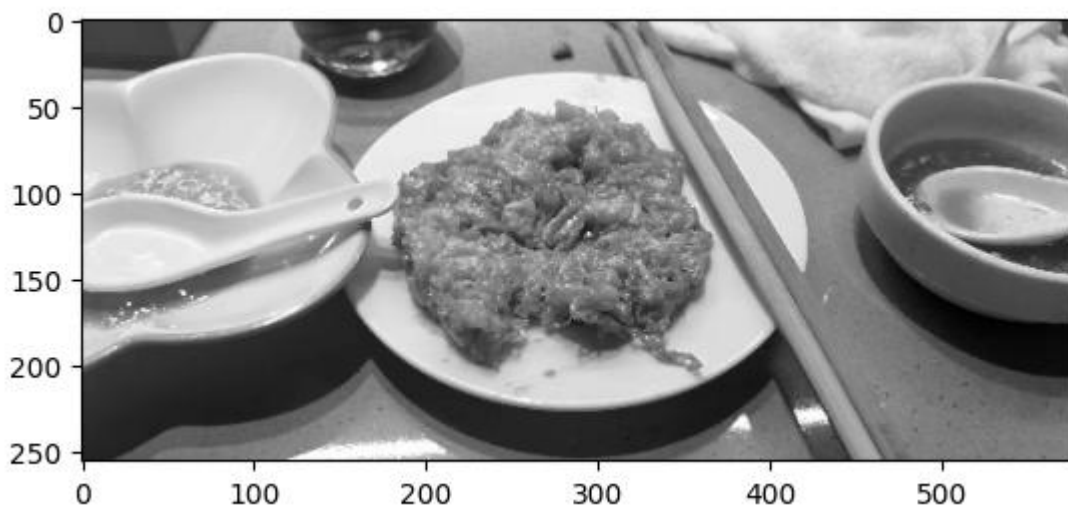
该函数的作用就是将图片池化, 池化的意思是将原图片尺寸进行压缩, 只保留重要特征。

before max pool, image shape: 1024 x 2310

after max pool, image shape: 256 x 577

可见图片尺寸变为了原来的四分之一。

效果图如下:



可以看到图片几乎没有变化, 说明池化层只是减小了图片的尺寸, 并不会影响图片的内容。

(二) 实验 2: 熟悉 batch normalization, 并实现 layer normalization

1. 熟悉 batch normalization, 对比使用归一化方法的效果:

使用归一化的训练结果:

```
Epoch 0. Train Loss: 0.314867, Train Acc: 0.910883, Valid Loss: 0.190516, Valid Acc: 0.942300, Time 00:04
Epoch 1. Train Loss: 0.171661, Train Acc: 0.950883, Valid Loss: 0.141973, Valid Acc: 0.958100, Time 00:04
Epoch 2. Train Loss: 0.132289, Train Acc: 0.960883, Valid Loss: 0.146328, Valid Acc: 0.956300, Time 00:04
Epoch 3. Train Loss: 0.111391, Train Acc: 0.967750, Valid Loss: 0.122683, Valid Acc: 0.962600, Time 00:04
Epoch 4. Train Loss: 0.096681, Train Acc: 0.971700, Valid Loss: 0.110802, Valid Acc: 0.967500, Time 00:04
Epoch 5. Train Loss: 0.083795, Train Acc: 0.975767, Valid Loss: 0.107857, Valid Acc: 0.968000, Time 00:04
Epoch 6. Train Loss: 0.076572, Train Acc: 0.977600, Valid Loss: 0.105257, Valid Acc: 0.967900, Time 00:04
Epoch 7. Train Loss: 0.068733, Train Acc: 0.979483, Valid Loss: 0.095048, Valid Acc: 0.971800, Time 00:04
Epoch 8. Train Loss: 0.060716, Train Acc: 0.982267, Valid Loss: 0.101903, Valid Acc: 0.970100, Time 00:04
Epoch 9. Train Loss: 0.055923, Train Acc: 0.983383, Valid Loss: 0.129603, Valid Acc: 0.962400, Time 00:04
```

不使用归一化的训练结果:

```
Epoch 0. Train Loss: 0.403578, Train Acc: 0.872417, Valid Loss: 0.253814, Valid Acc: 0.924200, Time 00:03
Epoch 1. Train Loss: 0.180932, Train Acc: 0.945183, Valid Loss: 0.173767, Valid Acc: 0.947700, Time 00:04
Epoch 2. Train Loss: 0.136090, Train Acc: 0.958250, Valid Loss: 0.156229, Valid Acc: 0.951600, Time 00:03
Epoch 3. Train Loss: 0.112475, Train Acc: 0.965267, Valid Loss: 0.117186, Valid Acc: 0.963100, Time 00:05
Epoch 4. Train Loss: 0.095188, Train Acc: 0.970667, Valid Loss: 0.101076, Valid Acc: 0.969400, Time 00:04
Epoch 5. Train Loss: 0.084131, Train Acc: 0.973950, Valid Loss: 0.180084, Valid Acc: 0.942100, Time 00:03
Epoch 6. Train Loss: 0.075468, Train Acc: 0.976200, Valid Loss: 0.089431, Valid Acc: 0.971800, Time 00:03
Epoch 7. Train Loss: 0.068493, Train Acc: 0.978317, Valid Loss: 0.123656, Valid Acc: 0.960100, Time 00:03
Epoch 8. Train Loss: 0.060801, Train Acc: 0.981400, Valid Loss: 0.102553, Valid Acc: 0.971000, Time 00:04
Epoch 9. Train Loss: 0.055851, Train Acc: 0.981850, Valid Loss: 0.090121, Valid Acc: 0.973000, Time 00:03
```

可以看到虽然最后的结果两种情况一样，但是如果我们看前几次的情况，可以看到使用批标准化的情况能够更快的收敛，因为这只是一个小网络，所以用不用批标准化都能够收敛，但是对于更加深的网络，使用批标准化在训练的时候能够很快地收敛

2. 实现 Layer Normalization（源码见附件 zip）

（1）MLP + nn.LayerNorm（MNIST，全连接）

```
Epoch 00 | Train 0.2558/0.9295 | Test 0.1381/0.9590
Epoch 01 | Train 0.1094/0.9677 | Test 0.0997/0.9707
Epoch 02 | Train 0.0780/0.9770 | Test 0.0901/0.9723
Epoch 03 | Train 0.0605/0.9826 | Test 0.0862/0.9742
Epoch 04 | Train 0.0474/0.9861 | Test 0.0742/0.9770
```

（2）CNN + nn.LayerNorm（在卷积后做 LN）

```
Epoch 00 | Train 0.2212/0.9367 | Test 0.0665/0.9788
Epoch 01 | Train 0.0596/0.9820 | Test 0.0448/0.9857
Epoch 02 | Train 0.0465/0.9855 | Test 0.0384/0.9873
Epoch 03 | Train 0.0377/0.9889 | Test 0.0375/0.9874
Epoch 04 | Train 0.0331/0.9898 | Test 0.0418/0.9866
```

我用官方 `nn.LayerNorm` 实现了按“每个样本的特征维度”做归一化：先对被规范化的最后若干维计算均值与方差，再做标准化并施加可学习的仿射参数（weight、bias）；与 `BatchNorm` 不同，它不依赖 batch 统计、没有 running mean/var，train/eval 行为一致。在 MLP 上我对隐藏层维度设 `LayerNorm(hidden)` 放在激活前；在 CNN 上我采用“通道维归一化”的常用写法，把特征图临时转成 NHWC，对 C 做 `LayerNorm(C)` 再转回 NCHW，避免受 H/W 变化约束。

（三）实验 3：VGG + 数据增强 + 正则化

1. VGG 跑通

（1）模型结构：

```

Sequential(
  (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU(inplace=True)
  (2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): ReLU(inplace=True)
  (4): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (5): ReLU(inplace=True)
  (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)

```

(2) 下面定义一个函数对这个 **vgg block** 进行堆叠:

```

def vgg_stack(num_convs, channels):
    net = []
    for n, c in zip(num_convs, channels):
        in_c = c[0]
        out_c = c[1]
        net.append(vgg_block(n, in_c, out_c))
    return nn.Sequential(*net)

```

(3) 定义一个具有 8 个卷积层的 **vgg** 结构:

```

Sequential(
  (0): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (1): Sequential(
    (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (2): Sequential(
    (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))

```

```

(1): ReLU(inplace=True)

(2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))

(3): ReLU(inplace=True)

(4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)

)

(3): Sequential(
  (0): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU(inplace=True)
  (2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): ReLU(inplace=True)
  (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)

(4): Sequential(
  (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU(inplace=True)
  (2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): ReLU(inplace=True)
  (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)

)

```

(4) 训练模型看看在 **cifar10** 上的效果

Epoch 0. Train Loss: 2.303178, Train Acc: 0.098545, Valid Loss: 2.303159, Valid Acc: 0.100969, Time 00:00:11
Epoch 1. Train Loss: 2.303034, Train Acc: 0.099724, Valid Loss: 2.302579, Valid Acc: 0.100969, Time 00:00:10
Epoch 2. Train Loss: 2.302891, Train Acc: 0.100543, Valid Loss: 2.302170, Valid Acc: 0.099585, Time 00:00:09
Epoch 3. Train Loss: 2.199893, Train Acc: 0.156730, Valid Loss: 2.044572, Valid Acc: 0.190763, Time 00:00:09
Epoch 4. Train Loss: 1.810127, Train Acc: 0.297854, Valid Loss: 2.198598, Valid Acc: 0.242286, Time 00:00:09
Epoch 5. Train Loss: 1.573864, Train Acc: 0.402334, Valid Loss: 1.544284, Valid Acc: 0.420293, Time 00:00:09
Epoch 6. Train Loss: 1.360163, Train Acc: 0.496603, Valid Loss: 1.496708, Valid Acc: 0.435819, Time 00:00:09
Epoch 7. Train Loss: 1.143138, Train Acc: 0.585018, Valid Loss: 1.217643, Valid Acc: 0.557555, Time 00:00:10
Epoch 8. Train Loss: 0.952949, Train Acc: 0.659607, Valid Loss: 1.288671, Valid Acc: 0.577037, Time 00:00:10
Epoch 9. Train Loss: 0.802796, Train Acc: 0.718430, Valid Loss: 1.307214, Valid Acc: 0.570609, Time 00:00:10
Epoch 10. Train Loss: 0.668370, Train Acc: 0.766524, Valid Loss: 0.792101, Valid Acc: 0.733386, Time 00:00:10
Epoch 11. Train Loss: 0.554402, Train Acc: 0.808304, Valid Loss: 1.391688, Valid Acc: 0.570510, Time 00:00:10
Epoch 12. Train Loss: 0.453936, Train Acc: 0.842771, Valid Loss: 0.931021, Valid Acc: 0.704114, Time 00:00:10
Epoch 13. Train Loss: 0.359291, Train Acc: 0.875899, Valid Loss: 0.722118, Valid Acc: 0.774229, Time 00:00:11
Epoch 14. Train Loss: 0.292835, Train Acc: 0.899896, Valid Loss: 0.807238, Valid Acc: 0.759098, Time 00:00:10
Epoch 15. Train Loss: 0.224279, Train Acc: 0.922275, Valid Loss: 1.285386, Valid Acc: 0.688489, Time 00:00:10
Epoch 16. Train Loss: 0.180049, Train Acc: 0.939478, Valid Loss: 0.956923, Valid Acc: 0.751384, Time 00:00:11
Epoch 17. Train Loss: 0.141580, Train Acc: 0.951626, Valid Loss: 2.380282, Valid Acc: 0.626681, Time 00:00:11
Epoch 18. Train Loss: 0.120150, Train Acc: 0.958879, Valid Loss: 0.983515, Valid Acc: 0.746440, Time 00:00:11
Epoch 19. Train Loss: 0.090304, Train Acc: 0.969389, Valid Loss: 0.896842, Valid Acc: 0.785305, Time 00:00:11

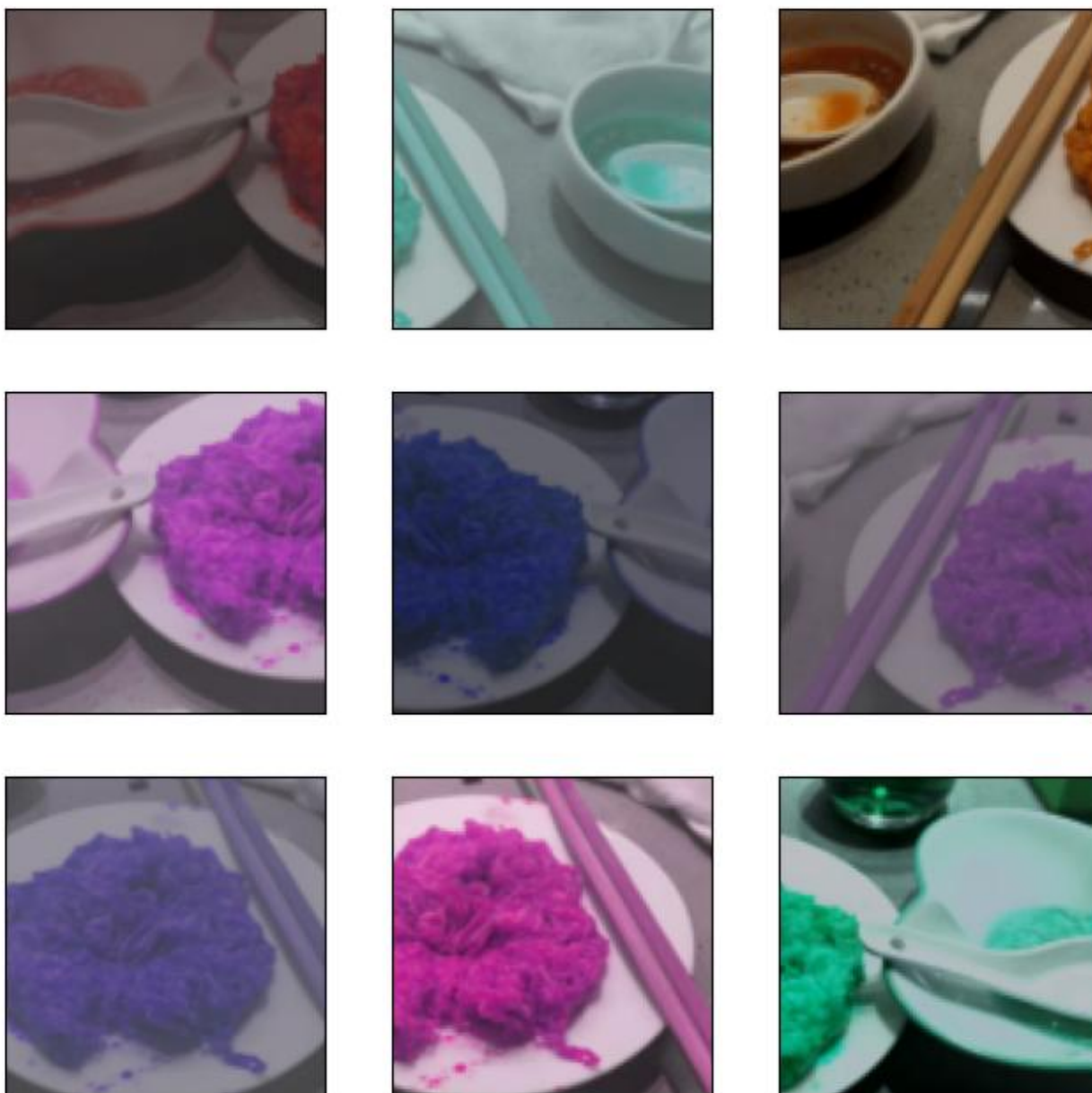
2. 数据增强

数据增强主要方法有：图片放缩、位置随机截取、随机的水平和竖直方向翻转、随机角度旋转、亮度对比度以及颜色调整。

（具体每一个的单独实现在此我就不展示了，附件中有，仅展示一个合辑。）

（1） torchvision 里面有个非常方便的函数能够将这些变化合起来

结果展示：



(2) 使用图像增强进行训练网络，看看具体的提升究竟在什么地方（使用 ResNet 进行训练）

1) 使用数据增强时：

Epoch 0. Train Loss: 0.079452,	Train Acc: 0.972846,	Valid Loss: 1.333593,	Valid Acc: 0.733188,	Time 00:00:09
Epoch 1. Train Loss: 0.067549,	Train Acc: 0.977482,	Valid Loss: 1.158564,	Valid Acc: 0.764636,	Time 00:00:10
Epoch 2. Train Loss: 0.059010,	Train Acc: 0.980039,	Valid Loss: 1.054932,	Valid Acc: 0.789161,	Time 00:00:10
Epoch 3. Train Loss: 0.048462,	Train Acc: 0.983875,	Valid Loss: 1.148177,	Valid Acc: 0.781151,	Time 00:00:10
Epoch 4. Train Loss: 0.041972,	Train Acc: 0.986873,	Valid Loss: 1.107191,	Valid Acc: 0.798062,	Time 00:00:10
Epoch 5. Train Loss: 0.038247,	Train Acc: 0.987432,	Valid Loss: 1.138123,	Valid Acc: 0.786689,	Time 00:00:10
Epoch 6. Train Loss: 0.038051,	Train Acc: 0.987912,	Valid Loss: 1.073351,	Valid Acc: 0.790348,	Time 00:00:10
Epoch 7. Train Loss: 0.031435,	Train Acc: 0.990309,	Valid Loss: 1.123235,	Valid Acc: 0.794699,	Time 00:00:10
Epoch 8. Train Loss: 0.028602,	Train Acc: 0.990829,	Valid Loss: 1.156781,	Valid Acc: 0.798161,	Time 00:00:10
Epoch 9. Train Loss: 0.015314,	Train Acc: 0.995404,	Valid Loss: 1.369874,	Valid Acc: 0.783327,	Time 00:00:10

2) 不使用数据增强时:

```
Epoch 0. Train Loss: 1.388974, Train Acc: 0.490549, Valid Loss: 4.126709, Valid Acc: 0.229233, Time 00:00:35
Epoch 1. Train Loss: 0.949543, Train Acc: 0.663043, Valid Loss: 1.390525, Valid Acc: 0.547271, Time 00:00:40
Epoch 2. Train Loss: 0.721418, Train Acc: 0.746943, Valid Loss: 1.075156, Valid Acc: 0.643493, Time 00:00:39
Epoch 3. Train Loss: 0.543598, Train Acc: 0.809783, Valid Loss: 0.980461, Valid Acc: 0.677512, Time 00:00:38
Epoch 4. Train Loss: 0.395974, Train Acc: 0.862712, Valid Loss: 1.741794, Valid Acc: 0.573279, Time 00:00:40
Epoch 5. Train Loss: 0.270152, Train Acc: 0.909287, Valid Loss: 1.115465, Valid Acc: 0.687203, Time 00:00:37
Epoch 6. Train Loss: 0.176264, Train Acc: 0.941696, Valid Loss: 1.096376, Valid Acc: 0.692840, Time 00:00:39
Epoch 7. Train Loss: 0.120479, Train Acc: 0.961217, Valid Loss: 1.965625, Valid Acc: 0.586036, Time 00:00:39
Epoch 8. Train Loss: 0.094937, Train Acc: 0.968790, Valid Loss: 1.142589, Valid Acc: 0.725771, Time 00:00:40
Epoch 9. Train Loss: 0.067629, Train Acc: 0.978041, Valid Loss: 1.082078, Valid Acc: 0.738034, Time 00:00:39
```

从上面可以看出, 对于训练集, 不做数据增强跑 10 次, 准确率到了 97%, 而使用了数据增强, 跑 1 次准确率就有 97%, 说明数据增强之后变得很简单。对于测试集, 使用数据增强进行训练的时候, 准确率也是会比不使用更高, 因为数据增强提高了模型应对于更多的不同数据集的泛化能力, 所以有更好的效果。

3. 正则化

正则化主要体现在三处: 一是**显式 L2 权重衰减** (weight_decay=1e-4), 在每步更新时惩罚过大的权重, 抑制过拟合; 二是**输入标准化** (Normalize([0.5]*3, [0.5]*3) 将像素缩放到 [-1, 1]), 虽然不直接减少参数, 但能稳定梯度、与 ResNet 中的 **BatchNorm** 协同, 起到**隐式正则**与训练稳态的作用; 三是**数据打乱与分批** (shuffle=True、中等 batch 大小) 带来噪声型正则化, 减轻对训练集顺序的记忆。

训练结果如下:

```
Epoch 0. Train Loss: 1.400205, Train Acc: 0.485734, Valid Loss: 1.519414, Valid Acc: 0.497528, Time 00:00:24
Epoch 1. Train Loss: 0.961828, Train Acc: 0.655531, Valid Loss: 1.154722, Valid Acc: 0.603837, Time 00:00:27
Epoch 2. Train Loss: 0.734582, Train Acc: 0.740749, Valid Loss: 1.204708, Valid Acc: 0.604826, Time 00:00:27
Epoch 3. Train Loss: 0.562014, Train Acc: 0.805527, Valid Loss: 1.193598, Valid Acc: 0.631230, Time 00:00:27
Epoch 4. Train Loss: 0.407533, Train Acc: 0.858776, Valid Loss: 1.199091, Valid Acc: 0.636966, Time 00:00:27
Epoch 5. Train Loss: 0.288707, Train Acc: 0.900955, Valid Loss: 1.123440, Valid Acc: 0.674150, Time 00:00:27
Epoch 6. Train Loss: 0.191728, Train Acc: 0.936081, Valid Loss: 1.084325, Valid Acc: 0.700158, Time 00:00:27
Epoch 7. Train Loss: 0.129567, Train Acc: 0.957581, Valid Loss: 1.470516, Valid Acc: 0.644778, Time 00:00:27
Epoch 8. Train Loss: 0.088173, Train Acc: 0.971727, Valid Loss: 2.487423, Valid Acc: 0.540348, Time 00:00:28
Epoch 9. Train Loss: 0.071827, Train Acc: 0.976543, Valid Loss: 1.089579, Valid Acc: 0.734276, Time 00:00:27
Epoch 10. Train Loss: 0.042029, Train Acc: 0.987072, Valid Loss: 1.014563, Valid Acc: 0.757911, Time 00:00:27
Epoch 11. Train Loss: 0.024276, Train Acc: 0.993606, Valid Loss: 1.220161, Valid Acc: 0.728639, Time 00:00:27
Epoch 12. Train Loss: 0.017346, Train Acc: 0.995744, Valid Loss: 1.105039, Valid Acc: 0.753659, Time 00:00:27
Epoch 13. Train Loss: 0.013151, Train Acc: 0.996943, Valid Loss: 1.006083, Valid Acc: 0.775514, Time 00:00:27
Epoch 14. Train Loss: 0.006420, Train Acc: 0.998841, Valid Loss: 0.996842, Valid Acc: 0.783525, Time 00:00:27
Epoch 15. Train Loss: 0.004014, Train Acc: 0.999520, Valid Loss: 1.424005, Valid Acc: 0.726562, Time 00:00:27
Epoch 16. Train Loss: 0.004865, Train Acc: 0.999181, Valid Loss: 0.964647, Valid Acc: 0.793216, Time 00:00:27
Epoch 17. Train Loss: 0.001815, Train Acc: 0.999820, Valid Loss: 0.971347, Valid Acc: 0.790348, Time 00:00:27
Epoch 18. Train Loss: 0.001262, Train Acc: 0.999940, Valid Loss: 0.979975, Valid Acc: 0.789458, Time 00:00:28
Epoch 19. Train Loss: 0.001210, Train Acc: 0.999940, Valid Loss: 0.973112, Valid Acc: 0.790546, Time 00:00:28
```

对比之下, 效果并没有显著提升, 说明正则化并不是很适用于我的图片。

四.实验总结

本次实验从“低层算子”到“整网训练”形成了连贯闭环：首先在`Conv2d`环节，基于拉普拉斯与 Sobel 卷积核对灰度/彩色图像进行边缘检测，通过调节`kernel_size/stride/padding`观察到边缘厚薄与输出尺寸的系统性变化，并用 min-max 归一化解决二阶/一阶导数响应的可视化对比度问题，建立了“参数—现象—机理”的直觉；随后以`MaxPool2d`验证局部极大下采样带来的平移不变性与计算开销下降，在尺寸压缩与信息保持之间取得平衡；在规范化部分，对比 BN 与 LN：前者依赖 batch 统计、训练收敛更快，推理用滑动均值/方差；后者对单样本按特征维归一化，batch 规模敏感度低、在小批量和序列/Transformer 场景更稳；接着搭建轻量 VGG（8 个卷积层 + 5 次池化）在 CIFAR-10 上训练，引入标准化的输入分布、基础数据增强（Resize/RandomCrop/Flip/ColorJitter/Normalize）与 L2 权重衰减形成轻量正则化组合，实验表明：增强显著提升泛化、L2 提升稳健性且与 BN 协同良好；综合来看，本实验明确了卷积/池化/规范化/增强/正则训练稳定性与泛化之间的分工与取舍，形成可复用的“小模型高性价比”实践范式，并为后续在更大数据、更深网络与更强数据增强（如 AutoAugment/混合增广）上的扩展奠定了方法论基础。